Internet Applications Design and Implementation

2015 - 2016 - 1st edition

(6 - Web Services)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Costa Seco (joao.seco@fct.unl.pt)
Jácome Cunha (jacome@fct.unl.pt)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

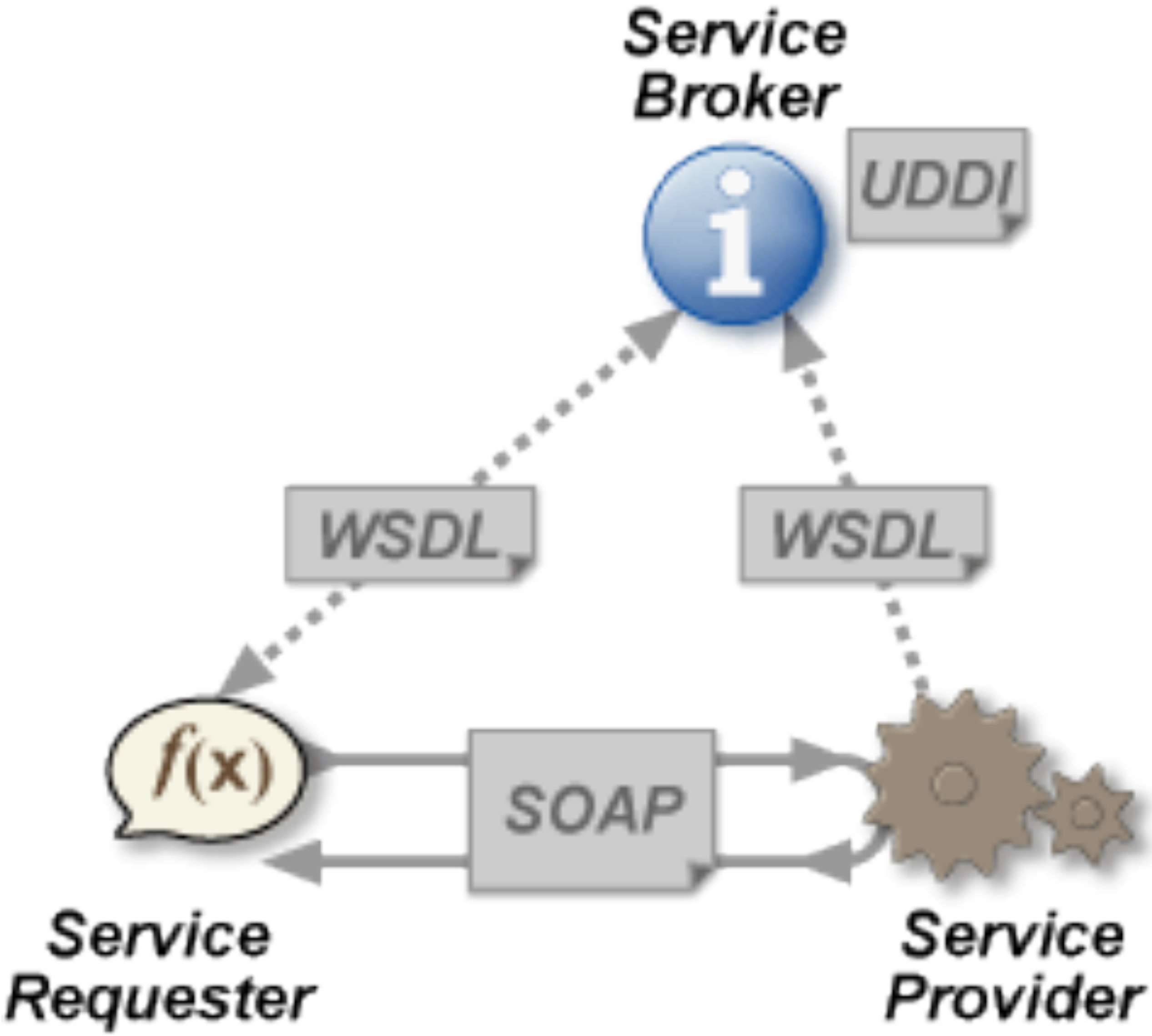
Web Services

Web Services - Basics

- Web services are web application components
- Can be published, found, and used on the Web
- They communicate using open protocols
- Are self-contained and self-describing
- Can be discovered using UDDI
- Can be used by other applications
- HTTP and XML is the basis for Web services

- By using Web services, your application can publish its function or message to the rest of the world

- Web services use XML to code and to decode data, and SOAP to transport it (using open protocols)


Service Broker
UDDI
WSDL
WSDL
f(x)
SOAP
Service Requester
Service Provider

Architecture

- The service provider sends a WSDL file to UDDI
- The service requester contacts UDDI to find out who is the provider for the data it needs
- Then it contacts the service provider using the SOAP protocol
- The service provider validates the service request and sends structured data in an XML file, using the SOAP protocol
- This XML file would be validated again by the service requester using an XSD file

2 Types of Use

- Reusable application-components
  - There are things applications need very often. So why make these over and over again?
  - WSs can offer application-components like currency conversion, weather reports, language translation, etc., as services
- Connect existing software
  - WSs can help to solve the interoperability problem by giving different applications a way to link their data
  - With WSs it becomes possible to exchange data between different applications and different platforms
  - Any application can have a Web Service component
  - WSs can be created regardless of PL

WSDL

WSDL

- WSDL stands for *Web Services Description Language*
- It is used to describe web services (no way!)
- Specifies the location of the service
- And the methods of the service
- Written as regular XML documents
- WSDL is a W3C recommendation since 2007

WSDL Documents

Element	Description
<types>	Defines the (XML Schema) data types used by the web service
<message>	Defines the data elements for each operation
<portType>	Describes the operations that can be performed and the messages involved
<binding>	Defines the protocol and data format for each port type

```
<definitions>

<types>
  data type definitions........
</types>

<message>
  definition of the data being communicated....
</message>

<portType>
  set of operations......
</portType>

<binding>
  protocol and data format specification....
</binding>

</definitions>
```

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

- The <portType> element defines "glossaryTerms" as the name of a port, and "getTerm" as the name of an operation

- "getTerm" operation has

  - an input message called "getTermRequest" and

  - an output message called "getTermResponse"

- The <message> elements define the parts of each message and the associated data types

The <portType> Element

- The <portType> element defines a web service, the operations that can be performed, and the messages that are involved
- The request-response type is the most common operation type, but WSDL defines four types:

Type	**Definition**
One-way	The operation can receive a message but will not return a response
Request-response	The operation can receive a request and will return a response
Solicit-response	The operation can send a request and will wait for a response
Notification	The operation can send a message but will not wait for a response

WSDL One-Way Operation

```
<message name="newTermValues">
  <part name="term" type="xs:string"/>
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="setTerm">
    <input name="newTerm" message="newTermValues"/>
  </operation>
</portType >
```

WSDL Request-Response Operation

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

WSDL Binding to SOAP

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
   <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http" />
   <operation>
     <soap:operation soapAction="http://example.com/getTerm"/>
     <input><soap:body use="literal"/></input>
     <output><soap:body use="literal"/></output>
  </operation>
</binding>
```

- The **binding** element has two attributes - name and type
    - **name**: (you can use any name you want) defines the name of the binding
    - **type**: points to the *port* for the binding, in this case the "glossaryTerms" port.
- The **soap:binding** element has two attributes - style and transport
    - **style**: can be "rpc" or "document". In this case we use document
    - **transport**: defines the SOAP protocol to use. In this case we use HTTP
- The **operation** element defines each operation that the portType exposes.
- For each operation the corresponding SOAP action has to be defined. You must also specify how the input and output are encoded. In this case we use "literal".

SOAP

SOAP

- SOAP stands for *Simple Object Access Protocol*
- It is an application communication protocol
- It is a format for sending and receiving messages
- It is platform independent
- Based on XML
- SOAP is a W3C recommendation since 2003

Why SOAP?

- It is important for web applications to be able to communicate over the Internet

- The best way to communicate between applications is over HTTP, because HTTP is supported by all browsers and servers

- SOAP provides a way to communicate between applications running on different operating systems, with different technologies and PLs

SOAP Building Blocks

- A SOAP message is an ordinary XML document containing the following elements:
  - **Envelope**: identifies the XML document as a SOAP message
  - **Header**: contains header information
  - **Body**: contains call and response information
  - **Fault**: containing errors and status information

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

Don’t change

Spring & SOAP Web Services

https://spring.io/guides/gs/producing-web-service/

REST

REST

- REST stands for *Representational State Transfer*
- It is a software **architectural style**
- **Not** a standard *per se*
- It may be implemented in different ways
- Systems that conform to the constraints of REST can be called **RESTful**
- RESTful systems typically, but not always, communicate over HTTP using its verbs (GET, POST, PUT, DELETE, etc.)

Architectural Properties

- **Performance** - component interactions can be the dominant factor in user-perceived performance and network efficiency
- **Scalability** to support large numbers of components and interactions among components
- **Simplicity** of interfaces
- **Modifiability** of components to meet changing needs (even while the application is running)
- **Visibility** of communication between components by service agents
- **Portability** of components by moving program code with the data
- **Reliability** is the resistance to failure at the system level in the presence of failures within components, connectors, or data

Architectural Constraints

- Architectural properties of REST are realized by applying specific interaction **constraints** to components, connectors, and data elements
- If a service violates any of the required constraints, it cannot be considered RESTful
- Complying with these constraints, and thus conforming to the REST style, enables any kind of system to have the desirable non-functional properties described in the previous slide

Architectural Constraints

- **Client–server**: a uniform interface separates clients from servers
- **Stateless**: client–server communication is further constrained by no client context being stored on the server between requests
- **Cacheable**: clients and intermediaries can cache responses
- **Layered system**: a client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way
- **Code on demand (optional)**: servers can temporarily extend or customize the functionality of a client by the transfer of executable code (e.g. JS)

Architectural Constraints

- **Uniform interface**: simplifies and decouples the architecture, which enables each part to evolve independently
  - **Identification of resources**: individual resources are identified in requests; resources are conceptually separate from the representations that are returned to the client
  - **Manipulation of resources through these representations**: when a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
  - **Self-descriptive messages**: each message includes enough information to describe how to process the message (e.g. MIME type, cacheability)
  - **Hypermedia as the engine of application state (HATEOAS)**: clients make state transitions only through actions that are dynamically identified within hypermedia by the server

REST & Web Services

- Web service APIs that adhere to the REST architectural constraints are called RESTful APIs
- HTTP-based RESTful APIs are defined with these aspects:
  - base URI, such as http://example.com/resources/
  - an Internet media type for the data; this is often JSON but can be any other valid Internet media type (e.g., XML, images, etc.)
  - standard HTTP methods (e.g., GET, PUT, POST, or DELETE)
  - hypertext links to reference state
  - hypertext links to reference-related resources

Example

RESTful API HTTP methods

Resource	**GET**	**PUT**	**POST**	**DELETE**
Collection URI, such as `http://api.example.com/v1/resources/`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as `http://api.example.com/v1/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it does not exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection.

Final Notes

- Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs

- This is because REST is an *architectural style*, while SOAP is a *protocol*

- Even though REST is not a standard *per se*, most RESTful implementations make use of standards such as HTTP, URI, JSON, and XML

Spring DEMO

Spring RESTful Web Service Example

```
@RestController
@RequestMapping(value="/hotelsrest")
public class HotelRestController {

    @Autowired
    HotelRepository hotels;

   // GET  /hotels       - the list of hotels
    @RequestMapping(method=RequestMethod.GET)
    public Iterable<Hotel> index(Model model) {
        return hotels.findAll();
    }

@RequestMapping(value="{id}", method=RequestMethod.GET)
    public Hotel show(@PathVariable("id") long id, Model model)
{
     Hotel hotel = hotels.findOne(id);
     if( hotel == null )
        throw new HotelNotFoundException();
     return hotel;
    }
```

Automatic Spring RESTful Web Service Example

```
@RepositoryRestResource(
     collectionResourceRel="hotelsautorest",
     path="hotelsautorest")
public interface HotelRepository extends
        CrudRepository<Hotel, Long> {
}
```